# 6.437 Project Part II

Robert Arnott

(Dated: May 8, 2018)

## OVERVIEW AND METHODOLOGY

In the first part of the project, I concluded by presenting a naive implementation of an algorithm which uses MCMC to evaluate the MAP estimator of a cipher, given encoded text and information about the english alphabet. Here, I will describe several technical challenges encountered in generating this estimate efficiently, as well as what I did to fix them. I stuck to the bigram model of the english language, and thus I will work with $\mathbf{P}$ and $\mathbf{M}$. Here, $\mathbf{P}$ provides a raw letter distribution, such that $\mathbf{P}_i = \mathbb{P}(x = a_i), a_i \in \mathcal{A}$, where $\mathcal{A}$ is the alphabet. $\mathbf{M}$ provides the bigram model, such that $\mathbf{M}_{ij} = \mathbb{P}(x_{n+1} = a_i | x_n = a_j), a_i, a_j \in \mathcal{A}$.

### Dealing with Small Probabilities

The first and arguably the most blatant problem with the naive implementation is that the probability of any sequence $x_1 x_2 ... x_N$, goes to 0 as $N \to \infty$. In particular, the length of the given ciphertext is large enough to where for basically all possible cipher functions, the probability of observing the ciphertext is below machine precision, at which point the system treats it as 0. Depending on how we handle the division $\frac{0}{0}$, this can lead or algorithm to terminate with an error, or make the accept probabilities 1 or 0. In order to fix this, I work exclusively with log-probabilities, which means I instead work with large negative numbers.

### Handling Zero-Probability Events

In most reasonable models of the english language, there is at least one zero-probability event. For example, I have that $\mathbb{P}(x_{n+1} = \text{“.”} | x_n = \text{“”}) = 0$ (the probability of a period following a space is 0), and $\mathbb{P}(x_{n+1} \neq \text{“”} | x_n = \text{“.”}) = 0$ (the probability of anything other than a space following a period is 0), among a couple others. Invariably, the algorithm will generate a sample cipher function which causes one of these events to occur in the decoded

test, thus assigning a zero likelihood to the sequence. This can cause several issues. To fix this, I handle 0 likelihood events by assigning them a small, non-zero probability. This allows the algorithm to progress through these samples, which it may have to in order to reach the true cipher function.

### Sub-Sampling

Calculating the log-likelihood of a long ciphertext can be quite expensive in terms of computation time. To get around this we instead generate a random subsequence from the ciphertext at each iteration and compute the accept probability using that subsequence. The optimal cipher function will of course assign the highest log-likelihood to all subsequences of the ciphertext, and by randomizing the subsequence seen at each iteration, I expect to see most of the ciphertext. This allows me to introduce a tradeoff between speed at each iteration, and correctness, by varying the length of the subsequence.

### Boosting

Because I enforce a limited number iterations, the algorithm is by definition, probabilistic. I can therefore increase the probability of finding an accurate cipher function by running it several times (i.e. by boosting it). I run the algorithm three times and choose the best estimator output by the 3. To prevent this from blowing up the runtime of the algorithm, I multi-thread the computation, so that the runs occur concurrently.

### Initialization

I first experimented with random initialization of the algorithm (i.e. I started at a randomly permutation of the alphabet and worked from there). Due to the heavy dependence on initialization of the speed of this algorithm, this would occasionally lead to poor results (making the idea of boosting even more important). In order to further reduce the chance of poor initialization and arguably improve the speed of the algorithm, I use the idea of frequency analysis (which was the first method used to solve substitution ciphers) to generate my starting sample. This starting sample is thus expected to be closer to the true cipher

function than it would be if I had randomly generated it.

## EVALUATION AND OPTIMIZATION

In addition to the performance tweaks mentioned above, I also spent some time optimizing a few things about my algorithm. In particular, since my algorithm lacks a probability-based stop criterion, I figured out how many iterations I should choose to run it for. I also had to find a good value for the sequence length used in subsampling (I chose one dependent on the length of the cipher text), and what value to choose for $\epsilon$ which controls the tradeoff embedded in my proposal (which is uniform over ciphers that differ in two assignments with probability $\epsilon$ and equal to the original state with probability $1 - \epsilon$).

For the number of iterations and sequence length, I grid searched over a discrete set of values. For example, for the number of iterations, I considered 1000, 2000, 3000,... iterations, while for the sequence length I tried $0.1N, 0.2N, ..., 0.9N$, where $N$ is the length of the ciphertext. I found that 3000 iterations on sequences of length $0.1N$ was ideal. For $\epsilon$, I chose $\epsilon = 1 - \frac{1}{\binom{m}{2}}$, where $m$ is the size of the alphabet. This makes the proposal uniform over the original cipher and all ciphers that differ in two assignments.